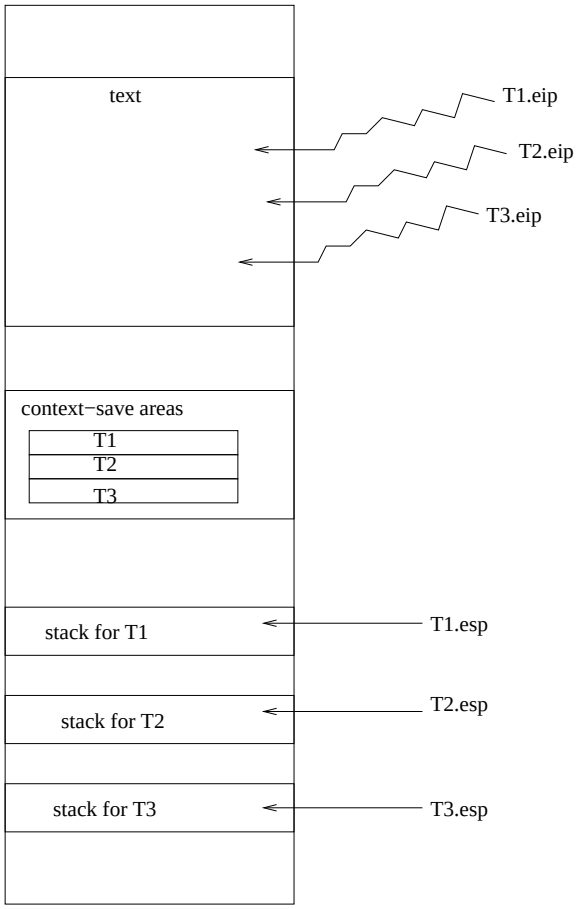
A Task Switch

Consider a multi-threaded user-level program. I.e. one address space, with multiple independent threads of execution within. What defines the **context** of execution on X86-32 architecture?



• The eip register determines the flow of execution (where the next instruction will be fetched).
• The eflags register contains the condition codes and thus affects how conditional branches will happen.
• The esp register controls calling/returning functions because it is where the return address is pushed/popped during CALL / RET. Each thread must have an independent non-overlapping memory area for stack, otherwise there is no meaningful way for functions to exist.
• The ebp register is used in many languages' run-time model to access local variables which live in the stack.
• Other registers such as eax, ebx, etc. are used to store working variables, intermediate results, etc.

So in other words, the values of the CPU registers define the context. Thus from a purely user-level standpoint, assuming that the operating system does not have any underlying support for multi-threaded processes, it would be possible for one task *voluntarily* to give away the sole (virtual) processor to another task by saving all of its registers in some safe place, and then restoring the other tasks' previously-saved register set into the cpu registers. Indeed, this user-level approach was used in very early versions of Linux before thread support was introduced (more than 20 years ago).

Preemptive and Cooperative Multitasking

In previous units, we have defined the terms Pre-emptive and Cooperative (or "Voluntary") multitasking. Since the Linux kernel is a controlled piece of code, it can be trusted to perform cooperative multitasking. However, user-mode programs can't be trusted to relinquish the processor. **Pre-emptive multitasking** means that a currently running task can be forcefully suspended and a context switch made to another, presumably ''better'' task to run.

The trigger for preemption may be based on priority, i.e. a higher-priority task has just become ready to run. It may also be based on time-slicing, in which each task is given a certain amount of time and then another task is given the CPU. This requires a **Periodic Interval Timer** interrupt. Or, it the trigger could be a blend of priority and time. In the Linux kernel, it is the `scheduler` subsystem which makes the decision as to which task to run and when.

In this unit, we shall study the implementation of multi-tasking in the Linux kernel. We have already seen that the kernel is a multi-threaded program of sorts where each thread (task) has its own kernel-mode stack. The situation therefore is analogous to the user-level voluntary task switch described above. The kernel is a *controlled* extension of a process's (task's) context into kernel mode. The kernel code, **acting on behalf of the process**, implements *cooperative* multitasking, which from the standpoint of user processes, appears to be *preemptive* multitasking.

In the Linux kernel, the term **task** is used to mean a schedulable thread of control. When processes are running programs which are multi-threaded, there are multiple tasks running around inside the same process address space. The Linux scheduler works on the basis of tasks, not processes.

The schedule() function

In the Linux kernel, a **task switch** takes place in the `schedule` function, which selects a new task to run (possibly the same task if the system is fairly quiet) and effects the switch

to the new task. Abstractly, `schedule()` is called and then another task has use of the processor. Then at some later time, the original task is scheduled again, and control appears to return transparently from `schedule()`. Internally, `schedule()` maintains a list of tasks which are in a ready to run state (as determined by the `state` field of the `task_struct`), and picks the "best" one. We call this list of ready tasks the **run queue**. (even though it is not strictly a queue)

`schedule` can be called in two ways:

• **Directly**: In a synchronous context, `schedule()` is called directly (or through an intermediate function or macro) when the current task wishes to relinquish (voluntarily) the processor because it has reached a blocking state. Examples include a system call which must block waiting for input, or a major page fault exception which must block the process until the page fault has been resolved by paging-in from backing store.

• **Indirectly (lazy)**: Whenever control is about to return from an interrupt, fault or system call, the value of the bit flag `TIF_NEED_RESCHED` in the `flags` word of the current `thread_info` structure is examined. We have seen this assembly language code in the previous unit when considering system calls, and similar code exists at the other "return to userland" points. If set, then `schedule()` is invoked. Therefore, when a kernel routine sets `TIF_NEED_RESCHED`, it is requesting that the currently executing task be (potentially) pre-empted and another task be scheduled. **An asynchronous handler must never call** `schedule` **directly**, because that would leave an unfinished interrupt handler pending. The task switch will occur upon return from the interrupt handler routine. Generally speaking, an interrupt handler routine will set the RESCHED flag if it is waking up a sleeping task which has higher priority than the current task, or when the timer interrupt routine determines that the current task has exhausted its cpu time slice quantum. (The RESCHED flag can also be set in a synchronous context)

Kernel Mode Pre-emption

On older Linux kernels, pre-emption could only occur when the task was about to return to user mode from an exception or interrupt handler. Modern Linux kernels can be built with support for **kernel mode pre-emption**, meaning that pre-emption can also occur when returning from the interrupt handler back to a kernel control path. Kernel pre-emption can improve latency performance in real-time applications, because it allows a higher priority task to run immediately. The `preempt_count` per-CPU variable controls kernel-mode preemption, and is examined when control is about to return from an interrupt handler back to kernel mode. The kernel macro `preempt_disable` increments this counter. Whenever the counter is non-zero, pre-emption is disabled. `preempt_enable` decrements the counter. Uses of `preempt_disable` can be "nested" and pre-emption is not re-enabled until the outermost kernel routine re-enables. In many places in the kernel, kernel pre-emption is temporarily disabled to prevent

leaving things in an inconsistent state.

Making a context switch -- overview (X86-32)

To make a task (context) switch on a given CPU, the kernel must:
• Ensure that all general-purpose registers (which might contain variable values) will be preserved. These can be saved on the kernel stack.
• Ask the scheduler which task should run next on that CPU
• Adjust scheduler parameters for the new and old task
• Save the stack pointer someplace other than the stack.
• Switch the stack pointer to the kernel mode stack for the new task
• Adjust the Task State Segment (TSS) so that the next time control re-enters the kernel from user mode, it does so onto the correct kernel mode stack.
• Adjust the current variable to point to the new task
• If applicable (switching from one user mode process to another vs just one thread to another) switch the user-mode Virtual Address Space by changing register CR3 to point to the new PGD.
• Save the current Program Counter (%eip register)
• And finally, update the Program Counter to "jump" to the previously suspended context point of the "new" task

Now that we've had this brief overview, we'll examine the kernel code in more detail to see the magic behind how these steps are performed.

What schedule() does

An vastly simplified outline of `schedule` is as follows:

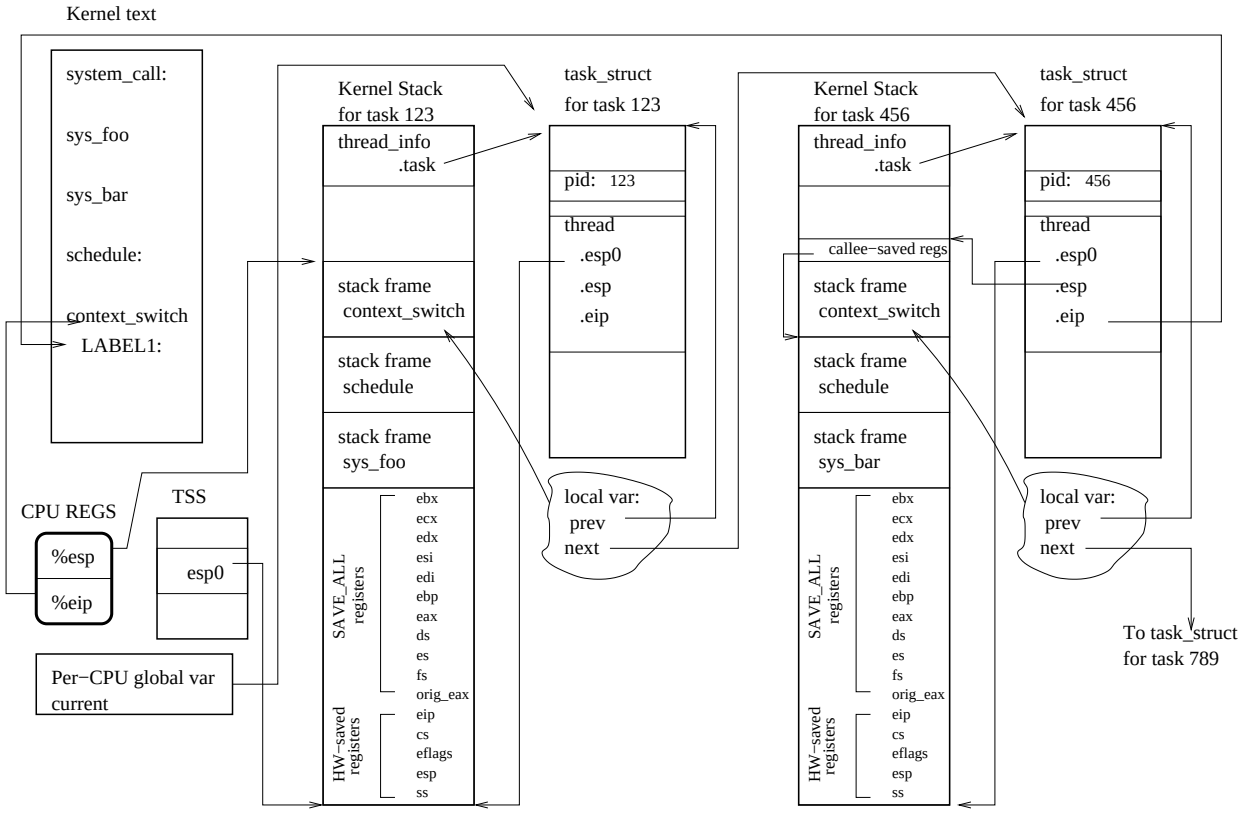
```
                /* from kernel/sched.c*/
schedule()
{
          task_t *prev,*next;
          runqueue_t *rq;

    need_resched:
          preempt_disable(); /* Temporarily disable kernel mode preemption*/
          cpu = smp_processor_id();    /* Get unique id of "this" CPU */
          rq=cpu_rq(cpu);              /* Run queue holding the current task */
          spin_lock(&rq->lock);        /* Lock against others modifying run queue */
          prev=rq->curr;               /* Remember the current task */
          /* ... update a bunch of scheduling statistics ... */
          /* pick_next_task is the "real" scheduler */
          next = pick_next_task(rq);
          prev->thread_info->flags&=~_TIF_NEED_RESCHED;   //Clear flag
          /* ... adjust run time stats of prev ... */
```

```
        if (prev!=next)             //Really going to switch now
        {
                sched_info_switch(prev,next);         /* Update timestamps, etc.*/
                rq->curr=next;
                 /* context_switch will also unlock rq*/
                context_switch(rq,prev,next);
                /* we are now re-scheduled, possibly on a different CPU */
                cpu = smp_processor_id();
                rq = cpu_rq(cpu);
        }
        else
                spin_unlock(&rq->lock);
        preempt_enable();
        if (current_thread_info()->flags & TIF_NEED_RESCHED)
                        goto need_resched;
}

context_switch(runqueue_t *rq, task_t *prev, task_t *next)
{
        prepare_task_switch(rq,prev,next);
        mm = next->mm;
        oldmm = prev->active_mm;
        switch_mm(oldmm,mm,next);             //switch address space

        /* Now it is time for the actual context switch.  This must be
           in assembly language.  The syntax below is simplified */
        asm(
                movl     prev,%eax
                movl     next,%edx
                pushl    %ebx               /* Preserve callee-save register */
                pushl    %edi               /* ebx, edi and esi, plus ebp */
                pushl    %esi               /* on prev's kernel stack */
                pushl    %ebp
 // THREAD is the offset of the .thread member of the task_struct
 // Likewise ESP is the offset of the .esp member of the thread struct
                movl     %esp,(THREAD+ESP)(%eax) /* prev->thread.esp=%esp */
                movl     (THREAD+ESP)(%edx),%esp      /* %esp = next->thread.esp */
 // At this moment we have switched to kernel stack of the NEXT task
                movl     $LABEL1,(THREAD+EIP)(%eax)   /* prev->thread.eip= LABEL1 */
 /* Pushing an address onto the stack and jumping to the function is
        the same as calling the function, but we will return to the
        pushed address instead of the next opcode */
                pushl    (THREAD+EIP)(%edx) /* push next->thread.eip */
                jmp     __switch_to         /* jump to C routine */
 /* When we are scheduled in again, we will come to life at LABEL1.  Yes,
        this is the next opcode.  Why did we bother?  We'll see when we
        look at fork! */
   LABEL1:
                popl     %ebp                          /* restore */
                popl     %esi
                popl     %edi
                popl     %ebx
                movl     %eax,prev          /* return val from __switch_to */
```

```
          );

          finish_task_switch(this_rq(),prev);             //Update stats
}


/* The fastcall directive says the arguments are in registers, not stack */
struct task_struct  fastcall *__switch_to(
                                    struct task_struct *prev_p,   //in eax
                                    struct task_struct *next_p)   //in edx
{
                    /* Get the thread_structs associated with old and new */
          struct thread_struct *prev= &prev_p->thread,
                               *next= &next_p->thread;
          int cpu=smp_processor_id();

                    /* Get the Task State Segment for this CPU */
          struct tss_struct *tss= &per_cpu(init_tss,cpu);
          tss->x86_tss.esp0 = next->esp0;                 //Set kernel re-entry stack ptr
          /* ... Deal with save/restore of additional registers which
                    are normally not touched in kernel mode, such as the
                    floating point, mmx/xxm/sse, debug registers, etc.
                    These are saved in the ->thread part of the task_struct */
          percpu_write(current_task,next_p);    //update current "variable"
          return prev_p;
}
```

Let us walk through a hypothetical example of making a context switch. Unfortunately, a problem with such examples is the circularity of reasoning. We'll assume that at some time in the past, task #456 had made a system call to (the hypothetical) `sys_bar()`, and that system call reached a blocking state, causing it to call `schedule()`. We will further assume that task #789 was selected by `schedule()` at that time and replaced task #456 on the CPU.

Now, some time has elapsed, and perhaps several intervening tasks have had use of the CPU. Task #123 is currently running in user mode. Somewhere along the way, task #456 was unblocked (e.g. because the input it was waiting for arrived) and thus was placed into the run queue. However, it has still not yet been scheduled. We now come to the beginning point of our example, when task #123 has made a system call `sys_foo()`, and `sys_foo` has to block task #123, thus it has called `schedule()` to yield the CPU. We assume that `schedule()` has selected task #456 as the best task to run next, and has just called `context_switch()`. The situation is depicted as follows:



Examining the kernel stack for task 123, we see the usual `thread_info` structure at the far end of the stack. We see the user's registers which were saved by the hardware and by the `system_call` kernel entrypoint code. We then see a stack frame created for `sys_foo()`, with the return address on the stack being the place where `sys_foo()` was called from the entrypoint code. Likewise, we see that `sys_foo()` called `schedule()`, and that `schedule()` (refer to the code listing above) has local variable `prev` pointing to task #123, and `next` pointing to task #456. Then `schedule()` has called `context_switch()` to effect the actual context switch. The CPU `%esp` stack pointer register is thus pointing to the next location in task 123's kernel stack. The `%eip` program counter register is pointing to the first instruction in `context_switch()`.

Now the following happens within `context_switch()`:

• We need to preserve the general-purpose registers. Recall from the appendix to Unit 7 that on the X86-32 architecture, registers ebx, edi and esi are "callee-saved" meaning that the compiler has generated code which expects that these registers will be safe after a CALL opcode (if the registers are "clobbered" by a called function, that "callee" needs to save and restore them). Since we are about to switch context, those registers would get "clobbered" and would not have the correct values when we are re-scheduled. Therefore it is up to `context_switch` to preserve them. The `%ebp` frame pointer register also needs to be saved. The top of the kernel stack **of task #123** is a safe place to stash these registers because nothing will touch it except task #123. Note that registers eax, ecx and

edx do not need to be saved because these are "caller-saved" registers and are not expected to survive the call to function `context_switch`.

• The current stack pointer (which points to the current top of the kernel stack for task #123, right after we saved the registers above) is saved in `prev->thread.esp` We can't save esp on the stack because how would we find the stack again after we change esp?!

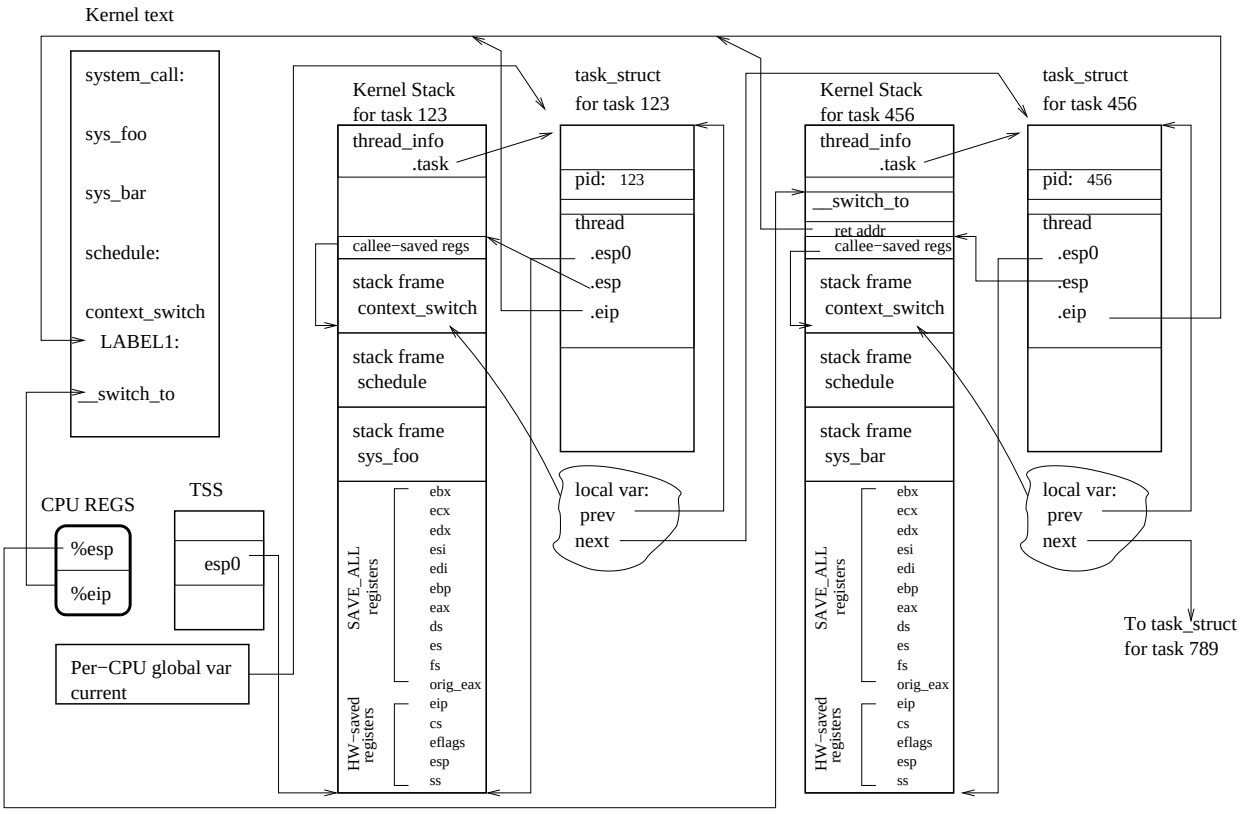
• The address which had been stored in `next->thread.esp` when task #456 was switched-out is also the stack pointer address corresponding to one word beyond the end of the stack frame for `context_switch()`, **BUT** within the kernel stack of task #456. This value is loaded into the `%esp` register, and therefore we are now on the stack of task #456.

• The address of the instruction labeled `LABEL1:` in `context_switch()` is stored in `prev->thread.eip`, i.e. in task #123's task_struct.

• The address which had previously been stored in the `thread.eip` field of task #456's task_struct is, likewise, that same address of the `LABEL1:` label. We'll see an exception to this when dealing with `fork()`. This value is fetched from `thread.eip` and pushed onto the kernel stack `of task #456`.

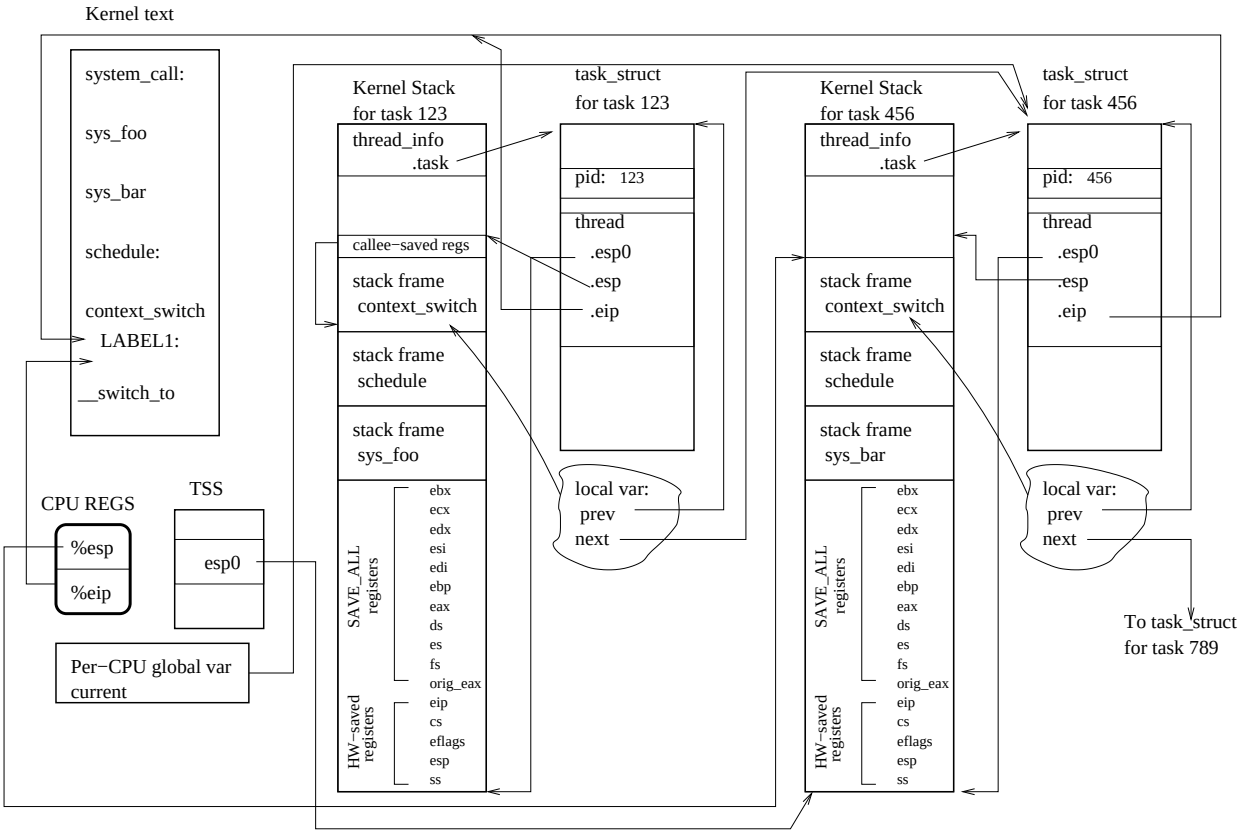
• We jump to the function `__switch_to()`. Normally, C functions are called with a `CALL` instruction, which saves the current `%eip` register on the stack as the return address. Here, the return address is the value which we had loaded from `next->thread.eip`, and manually pushed onto the stack. It so happens in this example that said address corresponds to the next instruction in `context_switch()`, i.e. label `LABEL1:`, but this mechanism allows us to resume execution when switched-in at some other place in the kernel code.

Now let's take a look at things when we have just entered `__switch_to()`:



`__switch_to()` updates the TSS so that at some later time, when task #456 is doing its thing in user mode, and control re-enters the kernel from user mode, it does so with the correct kernel stack pointer, i.e. the kernel stack of task #456.

When `__switch_to()` returns, its return value is the task which we replaced, i.e. the pointer to task struct #123. Thus the local variable `prev` in `context_switch()` is properly set. The `current` variable has likewise been set to point to task #456. We are now ready to return from `context_switch`, and then complete `sys_bar` and return to user mode in task #456:



Process and Thread Creation

Traditionally UNIX processes are self-contained virtual computers with their own private address space and operating system context (e.g. open file descriptors, signal handling). Under Linux, many aspects of the process have been broken out and it is possible to create a new process which selectively shares these attributes with the parent, by using the `clone` system call.

From user-level, the `clone` function is:

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

Unlike `fork`, `clone` begins execution in the child task in the function `fn` which is supplied with the argument `arg`. When this function returns, the child task exits. Before calling `clone`, an area of memory must be allocated and passed as `child_stack`. `clone` as described above is actually a user-level wrapper for the real `clone` system call:

```
int real_clone_syscall(int flags,void *child_stack,...)
```

Note that `fn` is not among these arguments. The real clone system call behaves like fork,

and the user-level wrapper makes sure that the child thread executes `fn(arg)`. The `clone` system call is rarely called directly by a program, and exists primarily to support the threads library (e.g. `pthread_create`). (Because this is an internal system call, the exact argument sequence has varied. What is shown here is a simplification. Programmers should never try to use this raw system call directly!)

Internal to the Linux kernel, there is no distinction between threads, lightweight processes, processes and tasks, and in fact these terms are often used interchangeably in the source code with confusing consequences. Each process has a unique process id. However, the POSIX Threads standard requires that all threads which exist within the same (heavy-weight) process have the same pid as returned by `getpid`. Therefore, Linux introduces the notion of `thread groups`. Each thread has a unique process id (`pid`), and all threads within a heavyweight process have a common `thread group id (tgid)`. It is the `tgid`, not `pid`, which is actually returned by the `getpid` system call. To get the value of the kernel-level `pid`, use the `gettid` system call (get thread id). For conventional single-threaded programs, the `tgid` and `pid` are of course identical.

`flags` contains the signal number to be sent to the parent when the child terminates (usually `SIGCHLD`). This is contained in the low-order byte. Other bitwise flags may be or'd in; an incomplete list follows:
• CLONE_VM: If set, share the virtual address space with the parent. If clear, the child has a copy-on-write private copy of the parent's address space at the instant of clone. If CLONE_VM is set, a new stack area must have been allocated and passed as the `child_stack` parameter, because if not, the child and parent would conflict in their use of the same stack at the same shared virtual address.
• CLONE_SIGHAND: If set, the parent and child will share the signal handling table. Any changes made by one will be reflected in the other's. If not set, the child gets a copy of the signal handler table in effect in the parent at the time of the clone.
• CLONE_FILES: If set, the parent and child will share the open file descriptor table. Therefore, any files open'd in one will be visible using the same fd number in the other. If not set, the child gets a copy of the file descriptor table at the time of the clone.
• CLONE_FS: If set, the parent and child will forever share the following file system information: current root of the filesystem (see `chroot`), current working directory, umask. If not set, the child gets a copy of the parent's information at the time of clone.
• CLONE_PARENT: Affects the notion of who is the parent process after the clone is complete, and thus affects `SIGCHLD` delivery. If set, the parent of the new child is the SAME as the parent of the caller. If not set, the parent of the new child is the caller.
• CLONE_THREAD: If set, the child is put into the same thread group as the caller, and therefore will have the same `tgid`. If not set, the child is placed in a new thread group of which it is the sole member and whose `tgid` is the same as the child's `pid`.

Typically, when `clone` is used to create a new thread with `pthread_create`, all of these CLONE_XX flags are set, so that the new thread exists within the same address

space, and any system calls made by a thread, such as opening a file, affect all of the other threads. Conversely, the `fork` system call sets none of the CLONE_XXX flags, and the new process is thus an independent copy.

```
/* Extremely simplified overview of struct task_struct */
/* For the purposes of illustrating fork/clone.  Missing a lot of stuff */
/* From /usr/include/linux/sched.h */
struct task_struct {
         struct list_head tasks;               // Part of linked list of all tasks
         struct mm_struct *mm;                 // Virtual memory layout
         int exit_code,exit_signal;   // exit status
         pid_t pid;                            // what gettid returns
         pid_t tgid;                           // what getpid returns
         struct task_struct *parent;  // who's your daddy?
         struct list_head children;   // linked list of child tasks
         struct list_head sibling;    // Part of linked list of siblings
         struct list_head thread_group;        // Part of ll of threads in thread grp
         cputime_t utime, stime;               // accumulated user/sys CPU time
         u64 start_time;                       // when this task was spawned
         struct cred *cred;
         char comm[TASK_COMM_LEN];    // Pathname of current executable
         struct fs_struct *fs;                 // cwd, umask, etc.
         struct files_struct *files;  // file descriptor table
         struct signal_struct *signal;         // Pending signals, etc.
         struct sighand_struct *sighand;       // Lots more signal stuff elided
};

struct task_struct *current_pointers[MAXCPU];   // Simplified
#define current (current_pointers[my_cpuid])    // Simplified

/********** (Simplified) Example of using the current pointer *******/
int sys_getpid()                      // getpid system call
{
         return current->tgid;
}

int sys_getuid()
{
         return current->cred->uid;
}
/*****************************************************************/
```

The `fork` and `clone` system calls both use the same underlying kernel function `do_fork()`to create a new task/process/thread:

```
/* The usual disclaimer: This code is a highly simplified extract
         of the actual Linux kernel source code */

int sys_fork()
{
         /* plain old fork is the same as clone with no CLONE_XXX flags */
         /* the lowest byte of clone_flags is SIGCHLD, which will be sent */
```

```
          /* to the parent when this new process exits.
          /* The child stack pointer is the same as the parent */
          /* since CLONE_VM is not set, the child gets a copy-on-write */
          /* private address space and thus it is OK that the SP are the same */
        return do_fork(SIGCHLD, 0);
}

int sys_clone(unsigned long clone_flags, unsigned long newsp)
{
        return do_fork(clone_flags, newsp);
);
```

Continuing to trace the flow of execution, both clone and fork call the same `do_fork` routine, which in turn calls `copy_process`:

```
long do_fork(unsigned long clone_flags, unsigned long stack_start)
{
        struct task_struct *p;
        int trace = 0;
        long pid = alloc_pidmap();              // Allocate new pid

        if (pid < 0)
                  return -EAGAIN;                       // Oops, out of PIDS
          /* A lot of simplification below */
        p = copy_process(clone_flags, stack_start, pid);
        if (!IS_ERR(p))
                  wake_up_new_task(p,clone_flags);      //place on run queue
          else
          {
                  free_pidmap(pid);
                  pid=PTR_ERR(p);               // Extract errno e.g. ENOMEM
          }
          return pid;
}
```

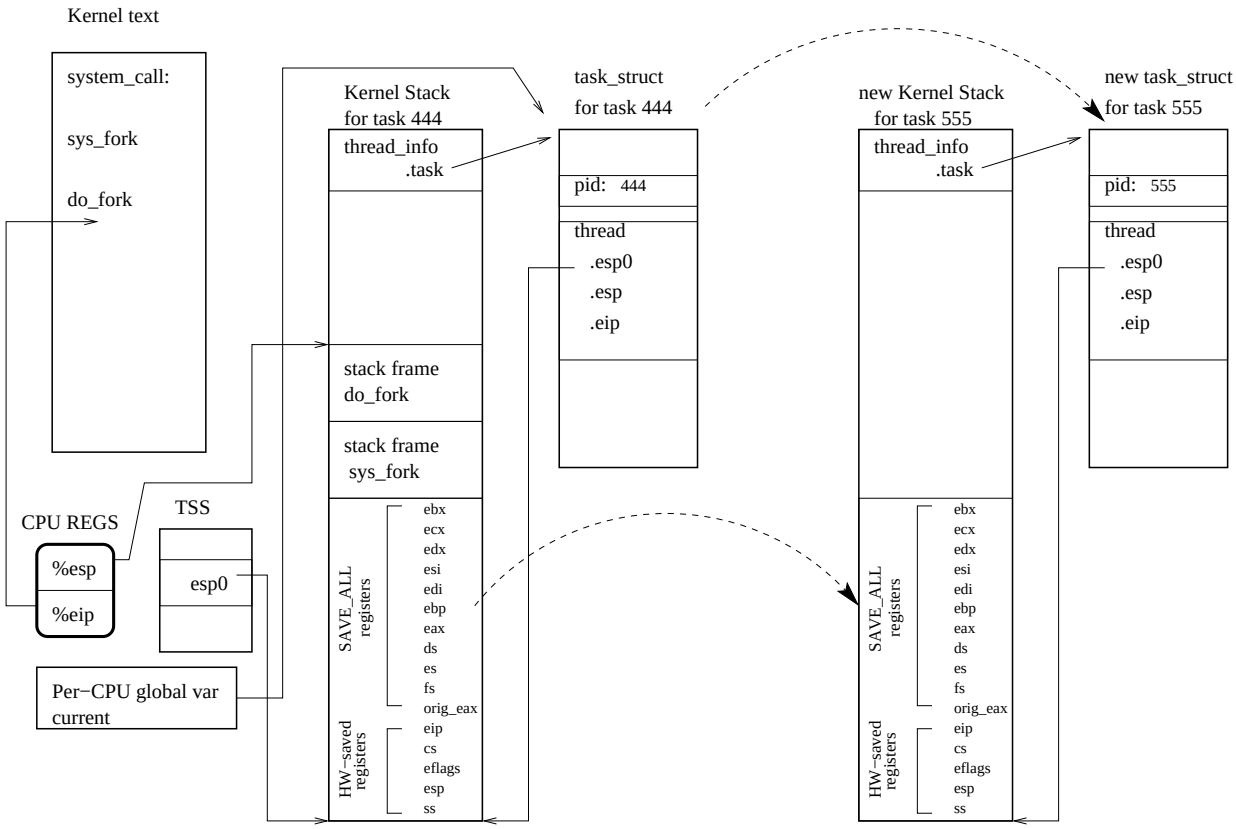
`do_fork` finds an available pid number for the new process or returns `EAGAIN` if there are no free numbers. `copy_process` is invoked, which allocates and returns a pointer to the new `struct task_struct` for the new process. Thanks to the range of valid addresses in kernel memory, a horrendously bad programming practice can be used within the kernel: the returned pointer is overloaded to contain either a valid pointer, or the (negative) error number, which will thanks to twos-complement be equivalent to a very high (and therefore invalid) pointer address. `copy_process` will return an error code if it did not succeed. Assuming all is well, `wake_up_new_task` is called to place the new task on a run queue so that it may be scheduled for execution.

The real meat is in the `copy_process` function:

```
static struct task_struct *copy_process(unsigned long clone_flags,
                                unsigned long stack_start,
                                int pid)
{
        int retval;
        struct task_struct *p = NULL;
```

```
            /* Some basic sanity checks of the clone flags */
  if ((clone_flags & (CLONE_NEWNS|CLONE_FS)) == (CLONE_NEWNS|CLONE_FS))
          return ERR_PTR(-EINVAL);
  if ((clone_flags & CLONE_THREAD) && !(clone_flags & CLONE_SIGHAND))
          return ERR_PTR(-EINVAL);
  if ((clone_flags & CLONE_SIGHAND) && !(clone_flags & CLONE_VM))
          return ERR_PTR(-EINVAL);
  retval = -ENOMEM;
    /* Make a shallow copy of the task_struct */
    if (!(p=alloc_task_struct())) goto fork_out;
    *p = *current;     /* Shallow structure copy */
    /* Allocate a new kernel stack, set cross-reference pointers */
    struct thread_info *ti;
    if (!(ti=alloc_thread_info())) {
              free_task_struct(p);
              p=NULL;
              goto fork_out;
    }
    p->thread_info=ti;
    ti->task=tsk;
    /* Because the task_structs are the same except for kernel stack addr,
     parent and child now share EVERYTHING.  The copy_XXX routines will
     break the sharing as requested */

    /* ... A bunch of stuff elided here to limit the number of processes
     which a user can spawn, etc. */

    p->pid = pid;
  p->tgid = p->pid;
  if (clone_flags & CLONE_THREAD)
          p->tgid = current->tgid;

  INIT_LIST_HEAD(&p->children);           // Set to empty list
  INIT_LIST_HEAD(&p->sibling);            // Set to empty list
    /* Establish empty signals pending for new task */
  clear_tsk_thread_flag(p, TIF_SIGPENDING);
  init_sigpending(&p->pending);

  p->utime = 0;
  p->stime = 0;
    acct_clear_integrals(p);     /* zero out per-process accounting
                        fields such as the #system calls, # page faults, etc. */
 /* Perform scheduler related setup. Assign this task to a CPU. */
  sched_fork(p, clone_flags);
  /* now copy all the process information */
    /* each of these copy_xxx functions will look at CLONE_XXX flags to */
    /* figure out if it is a share or a copy.  Some of the options */
    /* as well as the clean up for partial failure have been elided */
  if ((retval = copy_files(clone_flags, p)))       // open files table
          goto fork_out;
  if ((retval = copy_sighand(clone_flags, p)))     // signal handling tbl
          goto fork_out;
  if ((retval = copy_signal(clone_flags, p)))      // pending signals
```

```
                goto fork_out;
        if ((retval = copy_mm(clone_flags, p)))          // address space
                goto fork_out;
        copy_thread(clone_flags, stack_start, p);

          /* Mark new task to send signal on exit (normally SIGCHLD), only if
                     this is a new process, as opposed to a thread */
        p->exit_signal = (clone_flags & CLONE_THREAD) ? -1 : (clone_flags & CSIG
NAL);
        p->pdeath_signal = 0;
        p->exit_state = 0;
        if (clone_flags & (CLONE_PARENT|CLONE_THREAD))
                p->real_parent = current->real_parent;
        else
                p->real_parent = current;
        p->parent = p->real_parent;
                   /* Check to see if any signals came in during fork */
          recalc_signal_pending();
          /* ... Insert this process into the pid, pgid, tgid, etc. lists */
          retval=0;          /* We have succeeded! */
          total_forks++;              /* How many forks in life of system */
          nr_threads++;               /* How many current threads */
 fork_out:
          if (retval) return ERR_PTR(retval);
          return p;
}
```

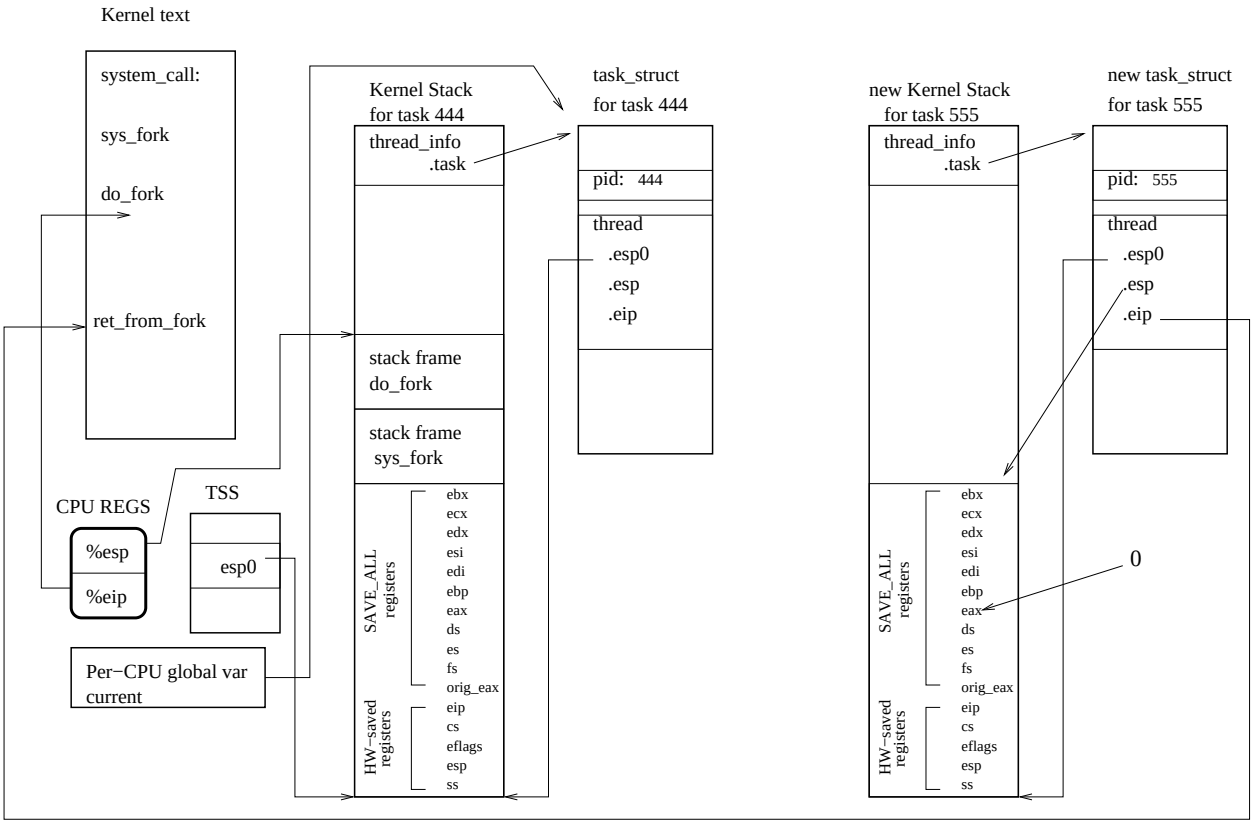


`copy_thread` initializes the brand-new kernel mode stack and task struct, which were created by `dup_task_struct`. The kernel stack for the child is initialized *as if* the child is in the process of making a system call. The registers which were stacked when the parent made the fork/clone system call are copied over, but a 0 is poked into the EAX register slot so that the child will see a 0 return value from the system call.

```
int copy_thread(unsigned long clone_flags, unsigned long stack_start,
        struct task_struct * child)
{
        struct pt_regs *childregs,*parentregs;
        struct task_struct *tsk;
        int err;

          /* Manipulate the user-mode register save area in the kstack */
        childregs = task_pt_regs(child);        // compute reg area address
          parentregs= task_pt_regs(current);
_info)) - 1;
          /* Copy all of the parent's user-mode registers to the child's kstack */
        *childregs = *parentregs;
                  /* Set the return value from fork/clone syscall in child to 0*/
        childregs->eax = 0;
                  /* Set the user mode stack pointer for when child first runs */
                  /* For clone, this will be a new value.  For fork, the
                           bulk register copy above set this to the parent's sp */
          if (stack_start)   // If child needs new stack VA
```

```
                childregs->esp = stack_start;

          /* Set the kernel mode stack pointer and stack base pointer */
          /* for when child is first scheduled */
        child->thread.esp = (unsigned long) childregs;
        child->thread.esp0 = (unsigned long) (childregs+1);
                /* When child is scheduled, execution will begin */
                /* in the return_from_fork assembly language function */
        child->thread.eip = (unsigned long) ret_from_fork;
          return 0;
}
```

Once `copy_process` and `do_fork` have completed their work, the child process is ready to run and is waiting on a run queue for its turn to get scheduled. We have already seen how an existing process gets scheduled in with `context_switch`. `do_fork` and its helper functions have now created a new process which is a copy of the current (parent) process (plus or minus the options specified with the CLONE flags) at the exact moment of returning from `fork` or `clone`. The situation just before the parent returns from `do_fork()` is:



Now, when the child process eventually gets scheduled, `context_switch` will find `ret_from_fork` as the new program counter location, rather than the usual address of a local label "LABEL1" within `context_switch`. This is required since the child is not being re-scheduled after having been switched-out, but is being scheduled for the first

time. The necessary stack frame for being in `context_switch()` does not exist in the child. Indeed, the only stack frame is the set of usual registers saved on entry to a system call. Therefore, the child comes to life at the following assembly language code:

```
ret_from_fork:
        pushl    %eax                          #contains prev task_struct ptr
        call     schedule_tail                 #schedule_tail(prev)
        GET_THREAD_INFO(%ebp)                  #mask esp to get thread_info to ebp
        popl %eax
        jmp syscall_exit            #resume exit from system call
```

`schedule_tail()` performs some cleanup actions related to the scheduler. Execution then continues at `syscall_exit` (see previous unit for listing), with the registers and stack looking like an ordinary return from system call. Control returns to user mode, with the return value of the system call being 0 (because the EAX slot in the user-mode saved registers on the child's kernel stack was poked with 0)

Scheduling

We have seen that the `schedule()` function is called to (potentially) make a task/context switch in the following situations:

• When the current task enters a non-READY state, e.g. because it makes a blocking system call.

• When the current task voluntarily yields the CPU, e.g. through the `sched_yield` system call.

• When the scheduler subsystem of the kernel decides that the current task's turn is done, and it is time for another task to have the CPU. This is known as **pre-emption**.

The Clock Tick

Regarding the last point, one specific interrupt is of great importance to scheduling. It is the **Periodic Interval Timer** or **'tick'** interrupt which arrives at a given frequency (every ms is typical). This gives the kernel the opportunity to pre-empt the task and give another task a chance to run, even if the current task does not make a system call or incur a fault and there is no other hardware interrupt activity on the system. The tick interrupt is also the timebase for the system time-of-day, things with timeout values such as the `alarm` system call and network protocols, etc.

When the clock tick interrupt arrives and control re-enters the kernel, the interrupt routine is able to determine if control came from user mode or kernel mode. It then "charges" a tick against the appropriate resource usage counter. In pseudo-code:

```
/* Make-believe kernel code*/
clock_intr_handler()
```

```
{
        if (came_from_user_mode)
                current->utime++;  // See above for meaning of current
        else
                current->stime++;
        clock_ticker++;             /* Increment global monotonic counter of ticks*/
        time_subsystem_tick();      /* Update time of day, run timeout
                                              callbacks, etc. */
        scheduler_tick();  /* Possibly switch tasks */
        /* Returns from interrupt */
}
```

Task Scheduling and Fairness

Under most circumstances, time spent executing on a CPU is a scarce resource for which tasks compete. This makes the **scheduler** part of the kernel an interesting problem that is often studied in OS research. Scheduling algorithms have varied widely from OS to OS, and even from one version to another. We can generalize a few broad principles:

• Tasks tend to be either **compute-bound** or **I/O-bound**. The former spend most of their time computing and thus have a heavy appetite for CPU time. The latter spend most of their time waiting for I/O. These classical definitions are often stressed by media applications, e.g. a streaming video server which is both I/O bound with network traffic and compute-bound with compression and decompression algorithms. Compute time among CPU-bound processes should be fairly distributed so that jobs complete in a reasonable time. Note that a given task may change its nature, e.g. a process such as Matlab which spends most of its time waiting for user input (I/O bound) but then has bursts of high CPU demand when it calculates results.

• Each task has an "importance", or **static priority**, which can be configured directly or indirectly by the system administrator to allow a task to receive a larger or smaller share of CPU time.

• The scheduler should allocate CPU time "fairly". Tasks that are at the same static priority level should, over a long sample period, receive approximately the same amount of CPU time. Tasks at different static priority levels should receive proportional amounts of CPU time.

• Tasks should appear as responsive as possible to interactive events. E.g. when a key is pressed or the mouse is moved, the application should respond quickly.

• The scheduler system itself should have a low overhead. The context switch is not a good time to be executing complicated, long-winded algorithms.

UNIX Static Priority Model

Historically, the static priorities in UNIX were represented by so-called `nice values`, ranging from -19 to +20, with a default value of 0. Positive "nice" values give a task

poorer static priority, i.e. they make it "nicer" to other, competing tasks with average (0) nice value. Classically, any task can increment its nice value, using the `nice` system call or command, but only a task running as the superuser (uid==0) can decrement the nice value and give itself "better" static priority. (In modern UNIX kernels, the privilege of giving oneself negative nice values is more fine-grained and can be assigned directly, without the process having to be the all-powerful, uid 0 superuser.)

Inside the kernel, other numbers may be used to represent priority, and they may have an entirely different interpretation from the traditional nice values. To maintain compatibility with POSIX standards, all kernels translate their internal priority number to the traditional -19..+20 nice value.

Static vs Dynamic Priority

To complicate matters, Linux uses the terms static and dynamic priority in a manner that differs from most authors on operating systems.

Traditional Definitions

• Dynamic priority: a relative value that fluctuates depending on what the task has done recently, and is consulted by the scheduler at each scheduling decision point (the "tick")
• Static priority: a fixed value that is established for the task and only changes by means of a system call. The nice value is an example of the traditional definition of static priorty. Note that the static priority influences the value of dynamic priority.

Linux Kernel Definitions

• Dynamic priority: the nice value, or other scheduling parameter that can be tuned.
• Static priority: one of 100 levels from 0 to 99. A ready task with higher static priority always runs before a task with lower priority. See the section on real-time scheduling.

The Quantum

A term used in the operating systems field regarding scheduling is **quantum** or "**time-slice**". This is the amount of time that a CPU-bound task runs before being pre-empted. Scheduling algorithms vary regarding their assignment of quantum. Some algorithms have an entirely fixed quantum, others a completely variable number, and others some intermediate solution (e.g. earlier Linux kernel schedulers used a variable quantum which was computed only when the task is scheduled in. Current kernels effectively re-compute the quantum at every tick)

Multi-processor systems and run queues

On a single-processor system, the list of all tasks which are READY to run is known as **the run queue**. It is not really a queue in the FIFO sense, because a task with better static priority can "jump the line" and get scheduled sooner than a poorer priority task that waits longer. So it is more of a rude queue with tasks cutting the line.

On a multi-processor system, there is one "run queue" for each processor. Each CPU has its own `current` variable and its own TSS. Typically, when a task becomes READY to run, the kernel decides which processor has the least workload and puts the task on that CPU's run queue. Thereafter, as long as the task remains READY, it typically remains on that CPU. This is because the cache for that CPU would still be "warm" with respect to the instructions and data that the task recently accessed. Under some circumstances, tasks can be migrated to a different CPU if loads get severely out of balance.

If the system is very quiet, it is possible that the number of READY tasks is less than the number of CPUs. In that case, some of the CPUs are assigned the **Idle Task**. This is either an endless loop waiting for something else to do, or an invocation of power-save mode which is un-done when a hardware interrupt arrives (including possibly an IPI to alert the idle CPU that a new task has been added to its run queue)

Normal Interactive Process Scheduling in Linux / CFS

The scheduling algorithm used for "normal" tasks by the Linux kernel is known as the Completely Fair Scheduler (CFS). It aims to satisfy the principles set forth above. The CFS algorithm attempts to provide "ideal latency" to all CPU-bound processes. Latency is defined as how much time elapses from when a task is pre-empted to when it gets the CPU again.

As a practical matter, the latency has a lower bound, because otherwise the system would spend most of its time in task switches, instead of doing useful work. Let us call P the latency period, and let us say this tunable value has been set to 10ms. If there are N=2 runnable tasks of equal static priority, each could run for 5ms and this would satisfy that the latency period, P, that each tasks sees should be 10mS. However, as N grows, this would imply smaller and smaller time slices, and eventually the overhead of scheduling and context switching will become prohibitively high. Another tunable parameter, G, is the scheduler granularity, the minimum time slice that tasks could have. If P/N<G, then P is capped at G. By default, in the Linux kernel, P is 5ms and G is 1mS, so if there are more than 5 runnable tasks (per CPU) then the latency period gets capped at 1ms. Since the clock tick is almost always 1ms, having a time slice of <1ms isn't possible anyway.

Weighted timeslice

The "nice" value, under the Linux CFS scheduler, is a process scheduling "weight". There are 39 nice steps (-20 to +19). Each nice step represents a 10% relative difference in CPU allocation. (This is a purely Linux interpretation of nice values -- other operating systems may have very different policies). The table below converts nice values into the weights, represented by the capital letter W:

```
/* nice -20 */ 86.6807,  70.0732,  55.1592,  45.1885,  35.4404
/* nice -15 */ 28.4707,  22.7090,  18.2666,  14.5986,  11.6367
/* nice -10 */ 9.3242,   7.4414,   5.9570,   4.7891,   3.8145
/* nice -5  */ 3.0479,   2.4424,   1.9443,   1.5488,   1.2471
/* nice +0  */ 1.0000,   0.8008,   0.6396,   0.5137,   0.4131
/* nice +5  */ 0.3271,   0.2656,   0.2100,   0.1680,   0.1338
/* nice +10 */ 0.1074,   0.0850,   0.0684,   0.0547,   0.0439
/* nice +15 */ 0.0352,   0.0283,   0.0225,   0.0176,   0.0146
```

Let's say process A has a nice value of 0 and process B has a nice value of 1, and these are the only two runnable processes. Then the weights (rounded off) are WA=1.00, and WB=0.8. We define the "load weight" LW as the sum of the weights of all runnable tasks. In this case, LW=1.800.

Next we define the CPU share for any task, w*n*, as w*n*=W*n*/LW. For A this is 1.00/1.800=55.5% and for B = 0.800/1.800 = 44.5%. Thus we see that the difference between two tasks separated by one nice level is approximately 10%. This formula is logarithmic, with each step in the table above being a multiplier of 1.25 relative to the next step. We see that if A had a nice value of -20 and B +19, A would get 99.98% of the CPU and B would get just 0.02%.

Under this weighted model, the perfect timeslice for a given task would be s*n*=P*w*n. Let us say P=10ms. A would have a weighted timeslice of 5.55msec, and B would get 4.45msec. Together they consume 10mS, the desired latency period.

This algorithm extends trivially to any number of runnable processes, and insures that the targeted weighted timeslice of any given process is its "fair share" of the available CPU, considering all of the other runnable processes and their weights. In actual kernel implementation, the weights are coded as integers, rather than the floating-point numbers used above, because the kernel avoids the use of the floating point registers.

Virtual Runtime

Under the CFS scheduler, the figure of merit when comparing runnable processes for scheduling is unfortunately called *virtual runtime*, or vruntime. Like many things in the Linux kernel, this is poorly named, and perhaps would better be called the "weighted actual runtime share".

The idealized allocation of timeslice presented in the previous section can not be realized in practice because 1) pre-emption only happens during a scheduler tick, which has a granularity of (typically) 1 msec, and 2) while a task is running, other tasks awaken, changing the weighted load.

At every scheduler tick, the ideal timeslice for the currently running process is recomputed. If the process has now been on the CPU for longer than that timeslice (subject to rounding to the nearest clock tick) then it is a candidate for re-scheduling. Re-

scheduling can also occur when the system load changes (because processes go to sleep or wake up, or because nice values are changed).

The vruntime of the running process (`current`) is updated: vruntime+= T/w, where T is the amount of time elapsed since the last time the load was examined (typically since the last clock tick) and w is the relative weight of the process (Wn/LW). The higher the relative weight, the less vruntime will be "charged" to the current process. The vruntime is cumulative for the life of the process.

Therefore, the lower the vruntime of a process, the greater is its relative merit for being scheduled now. The process with the lowest vruntime of all the runnable proceses is the one which should be on the CPU.

The CFS scheduler maintains a data structure (it is implemented as a red-black binary tree with caching of the lowest element) to keep all of the runnable tasks in order by vruntime. Retrieval of the "next best" task to run is therefore constant time. When the scheduler considers re-scheduling, it examines the tree for the task with the lowest vruntime. If the currently running task happens to be the lowest, then nothing happens, otherwise a task switch takes place and a new task gets the CPU.

```
/* Vastly simplified pseudo-code presentation of actual Linux kernel code */
scheduler_tick()
{
          /* Assume CLOCK_PERIOD is 1ms */
          current->vruntime += CLOCK_PERIOD / current->w;
          ideal_timeslice = P * current->w;
          if (clock_ticker - current->scheduled_in_ticktime  > ideal_timeslice)
          {
                    t = find_best_runnable_task();
                    if (t->vruntime < current->vruntime)
                              context_switch-to(t);
          }
          /* At this line of code, current is the next task to run,   *
           * which may be the same task as before or may be a new one */
          current->scheduled_in_ticktime = clock_ticker;
}
```

When examined in fine-grain detail, the actual timeslices of tasks will not match their "ideal" computed value. But on the average over a longer period of time, the selection of the next task to run based on lowest vruntime will result in an equitable distribution of CPU time which is approximately equal to that which would have been obtained if it were possible to implement the ideal weighted timeslice. This is depicted below, with tasks A and B as previously described having nice values of 0 and 1. The ideal timeslices

of 5.55 and 4.45 msec must be rounded up to 6 and 5, because it is impossible to give a task a timeslice which is a fraction of the tick time (we round up, rather than to the nearest integer, because a time slice of 0 is impossible).

When A is running, at every scheduler tick, it is assessed 1 msec / 0.555 = 1.8 units of vruntime, but when B is running, it gets charged for 1 msec / 0.445 = 2.25 units of vruntime. Looking at the skeleton code for `scheduler_tick()` above, the kernel will not pre-empt the task until it has run out its ideal timeslice (this is a bit of a fib but accept it for now). If no other tasks become ready, A will continue to get 6ms slices and B 5ms. After A has run for its 6 ticks, it has been charged 6 / 0.555 = 10.8 units, and after B runs for its 5 ticks, it is charged 5 / 0.445 = 11.2 units. The two tasks will alternate running for their respective timeslices, but each time A is being cheated of 0.55 msec and B is getting 0.55 msec bonus time. Eventually, the vruntime catches up with B and it skips one turn. For the purposes of the example timeline below, let us assume that the vruntimes of A and B start out at 1000.

Current	VRA	VRB	Action
n/a	1000	1000	A runs first
A	1010.8	1000	B selected
B	1010.8	1011.2	A selected
A	1021.6	1011.2	B selected
B	1021.6	1022.4	A selected
A	1032.4	1022.4	B selected
B	1032.4	1033.6	A selected
A	1043.2	1033.6	B selected
B	1043.2	1044.8	A selected
...			
B	1270	1280	A selected
A	1280.8	1280	B selected
B	1280.8	1291.2	A selected
A	1291.6	1291.2	B selected
B	1291.6	1302.4	A selected
A	1302.4	1302.4	A selected
A	1313.2	1302.4	B selected
B	1313.2	1324.8	A selected
etc.			

A and B alternate 6ms and 5ms time slices. After 27 volleys, A’s vruntime winds up being equal to B’s after running for its 6ms. Since B is not "better" then A gets another turn to run. A has run for 28 turns at 6ms each or 168ms of total actual CPU time, while B had 27 turns of 5 ms each or 135ms. In this 303ms elapsed period, A’s CPU share is 55.5% and B’s is 44.5%, just as it should be.

The `vruntime` value meets the traditional definition of dynamic priority, even if the Linux kernel doesn't use that term to describe it.

User vs System CPU Time

Some system calls may require a non-trivial amount of CPU cycles to complete. For example, a read from the char special device `/dev/urandom` requires the kernel to iterate a pseudo-random number generator function for as many bytes are requested. This usage of CPU resources is properly accounted for by the scheduler, because the task is "charged" for CPU time whenever the clock interrupt arrives and the task has the CPU. The two counters `utime` and `stime` keep track of which is which.

However, in order for the CPU usage to be "charged" to the process, it has to happen in a synchronous context. In some cases, the process is doing things which are causing high kernel-mode CPU usage, and it is not that easy to charge back to the process and to control with the scheduler. For example: a process that is doing a lot of network communication using an encrypted protocol where the encryption is handled in the kernel. This, and other "fairness" issues such as controlling disk I/O and network bandwidth usage by processes, continues to be an active area of OS research.

Scheduler interactions with fork

In the CFS scheduler, upon a fork, the child process inherits the vruntime (and nice value or weight) of the parent at the time of the fork. This somewhat mitigates the cheating that could otherwise occur if child processes were given a 0 initial vruntime.

Interactive performance & pre-emption

The vruntime approach naturally favors a process that has woken up after a long sleep, because its vruntime has not been incremented. When a process W is awoken, its vruntime is compared to the current process C. If VR(W)>VR(C), then the current process is still "better" than the awoken, and it will not be pre-empted. This can happen if the current task has a much better nice value than the awoken task.

The actual algorithm in the kernel is somewhat more sophisticated, because of the need to consider the last CPU that a task ran on before sleeping or getting pre-empted. Depending on how the load is balanced, and the length of time elapsed, it may be better to let a task wait a little longer in order to get back on its last CPU.

Group Scheduling

In a sense, it might be considered unfair that under the traditional UNIX scheduling model, a user who has 20 processes running is getting a bigger share of the overall CPU time than a user with just 1 process. The only cap to this is the per-user process limit. Modern Linux kernels allow for "group-based" scheduling. Tasks can be placed into groups (not related to the "group" as in gid), either automatically by uid, or manually by the system administrator. The percentage of CPU time allocated to each group can then be tuned by the administrator. Under this model, each user can be restricted to a maximum amount of the overall CPU time, and that user's tasks compete among themselves to divide up that share.

Real-Time and Quasi-Real-Time Scheduling

The scheduler in a general-purpose operating system is designed to provide fairness and interactive responsiveness. This is not necessarily appropriate for embedded systems which are controlling physical systems. Consider an absurdly hypothetical system which runs a nuclear power plant. Task A operates the control rods and task B updates power production records for billing purposes. If Task A becomes ready to run because some action needs to be taken with the control rods, it would not be a great design if it had to wait until Task B completed its billing computations.

In a real-time scheduling system, tasks are given fixed, static priorities by the system administrator. A high-priority task, if ready to run, always pre-empts a low-priority task. In fact, the lower priority task will not be able to run at all until all higher priority tasks are asleep. If multiple tasks at the same priority level are ready to run, some real-time schedulers say that the first task to become ready runs first and continues to run until it sleeps or voluntarily yields. Linux calls this `SCHED_FIFO` scheduling policy. Another approach, which Linux calls `SCHED_RR`, says that each ready real-time task in a given priority level will run for a fixed **quantum** or **time-slice**, after which the next ready task at that priority level will be allowed to run, etc. Eventually, the CPU will get back to the first task. This is known as **round-robin** scheduling. However, the pre-emption by higher priority tasks continues to take place.

The Linux kernel uses the convention that static priorities run from 0 to 99, with 99 being the best. Non-realtime tasks have static priority of 0, while realtime tasks have a non-zero static priority. Therefore, if *any* realtime task is ready to run, the non-realtime task has to wait. Only privileged users can create real-time tasks, otherwise it would be trivial to monopolize CPU resources.

The Linux kernel and most other general-purpose kernels are not truly real-time kernels, because they can not guarantee a specific minimum response time between an event (e.g. an interrupt arrives) and the scheduling of the real-time task. A term that is often used is "quasi" real-time or "nearly real time." There are kernels which are designed from the ground-up to be truly real-time. Such kernels are seen only in embedded systems and would not be very suitable for general-purpose computing. Often a real-time, embedded

system will use one kernel for the real-time stuff, and host a general-purpose kernel such as Linux for administration or user interface. However, Linux has been making steady progress. Recent kernels include support for deadline-based scheduling rather than simplistic static priority scheduling for realtime tasks.

I/O Scheduling and Fairness

Traditional UNIX systems only concerned themselves with CPU time allocation fairness. But in terms of user and administrator perceptions, the issue of disk I/O and network fairness is just as important. Think about an I/O bound system where user 123 has spawned off a bunch of processes that are making a lot of disk I/O requests. These processes have ordinary nice value of 0 giving them "average" priority. Now uid 0 has a system maintenance process with nice value of -20 (best priority). The nice value would only affect CPU timeslice allocation, but the scarce resource here is disk I/O bandwidth.

Modern Linux kernels solve this by using similar algorithms for determining I/O scheduling and network traffic scheduling, which is calls Completely Fair Queueing. Processes have an I/O priority which is analogous to the cpu scheduling `nice` value. When a given disk device has multiple pending requests, the I/O scheduler takes into account the I/O priority associated with the task that originated the I/O request and tries to apportion the I/O bandwidth accordingly, just as the CFS scheduler attempts to apportion CPU time. Similar mechanisms exist for fair allocation of network bandwidth and other I/O resources.

Sidebar: linked lists

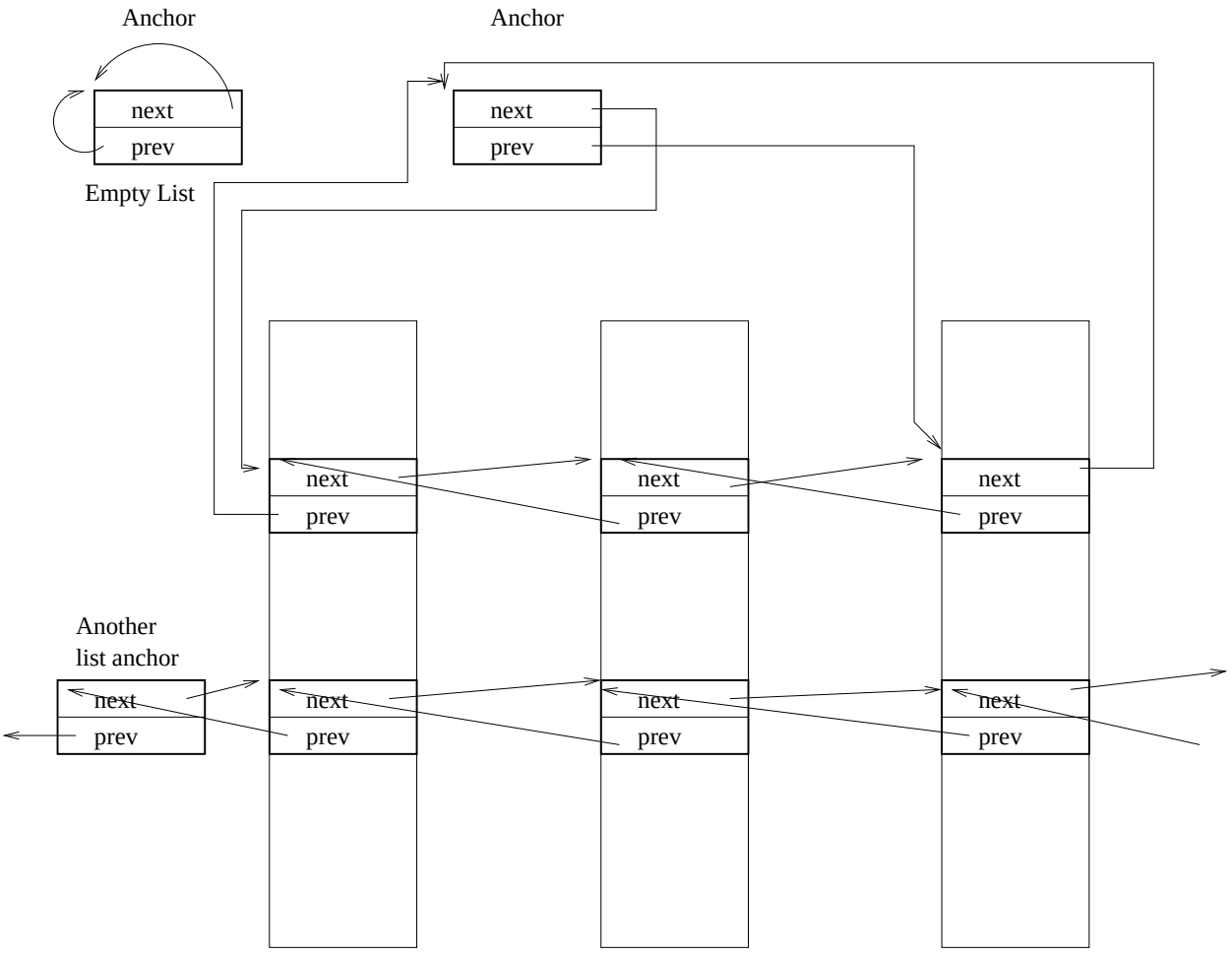
Within the Linux kernel, a frequently seen data structure is the `list_head`. The `struct list_head` contains two pointers `next` and `prev` to implement the doubly-linked list. However, rather than pointing at the element itself, these pointers point to the `struct list_head` within each element. The result is that generic list manipulation routines can be used for any list of type FOO, because the offset of the next and prev pointers within an element of FOO is not needed. The macro `list_entry` gets back to the element of type `type` in which `ptr` is the address of the list_head field named `member` within the entry.

```
#define list_entry(ptr, type, member) (                       \
        (type *)( (char *)ptr - (size_t) &((type *)0)->member))
```

Furthermore, the doubly-linked list is circular, meaning there are never NULL pointers. This is advantageous in that it removes a lot of conditional code. The entire list is also represented by a `struct list head`, also known as the list *anchor*. If we call this

anchor A, then `A.next` points to the first item on the list, and `A.prev` is the last. It is therefore just as easy to insert an item at either end of the list, which is again used to advantage in many places. The anchor itself is not considered an element of the list. Therefore an empty list is represented by having an anchor where both `next` and `prev` point back to the anchor.

It is frequently the case that a given `struct FOO` inside the kernel contains more than one `list_head` member. Each threads a different circular, doubly-linked list through FOO. E.g. the `task_struct` is on a list of all tasks in the system, and also a distinct list of all children of a particular task, etc.



Sleep and wakeup

Frequently within the kernel a situation is encountered in which a task, executing in a synchronous context, must wait for an event to occur. Examples include:

- System call which needs to wait for an I/O operation to complete.
- System calls involving reading from an empty pipe, or writing to a full pipe.

• Reads from a socket with no data pending, or write to a socket with the buffer full.
• System calls such as `wait` which wait until the state of another task changes.
• "Major" page faults which block until a disk I/O operation completes.

Sleeping on an event and waking up when it occurs involves two context switches, one voluntary, the other potentially pre-emptive. The first, voluntary switch comes when say task "A" encounters the blocking condition in a synchronous context. It places itself on a wait queue, as described below, and then calls `schedule()` voluntarily to yield.

At some later time, the event is satisfied, either in an asynchronous context (e.g. I/O complete interrupt) or a synchronous context (e.g. another process writes to a pipeline). This causes task "A" to be marked as ready to run. A context switch may be forced if the task running at that time, say task "Y", has inferior priority to task "A". Or, a context switch may happen when task "Y" uses up its time slice (see unit 10). In these two cases, the context switch is pre-emptive, and occurs via the `TIF_NEED_RESCHED` flag when task "Y" returns to user mode. Or task A could be scheduled if task Y enters the kernel and blocks.

Wait Queues

For each waitable event, a kernel data structure known as a **wait queue** is defined. It is implemented as a circular, doubly-linked list with some unusual tricks which improve efficiency. This list chains together all of the tasks which are waiting on a specific event.

The wait queue is anchored at a wait queue head structure:

```
typedef struct __wait_queue_head {
         spinlock_t  lock;                    /* mutex */
         struct list_head task_list;          /* anchor */
} wait_queue_head_t;
```

Each element of the list is:

```
typedef struct __wait_queue {
         unsigned int flags;
         struct task_struct *task;
         int (*func)();                                  /* callback */
         struct list_head task_list;          /* chains */
} wait_queue_t;
```

A wait queue of type `wait_queue_head_t` contains a circular, doubly-linked list of `wait_queue_t` entries, each of which represents one task waiting on that event.

Kernel code which places tasks on a wait queue is executed in a synchronous situation, because it is running in the context of a process which is being blocked. On a multiprocessor system, multiple kernel routines could be executing in true parallel and potentially trying to insert into the same wait queue (e.g. picture several processes simultaneously blocking on a read from the same input source.) Since this operation is not inherently atomic the `spin_lock` element of the list anchor must be used to protect

it.

One might ask "why is a spin lock being used to protect the list head, rather than a blocking mutex or semaphore?" The reason is (A) the time that kernel routines are in the critical region is brief and bounded....they need only to obtain the spin lock, manipulate a few pointers to insert a wait queue entry, then release it. (B) Furthermore, code which wakes up tasks is often called from an asynchronous context, e.g. after an I/O completion interrupt. Therefore it can not perform a blocking operation, because it would be blocking a process which is in effect an innocent bystander and which has nothing to do with the wait queue in question.

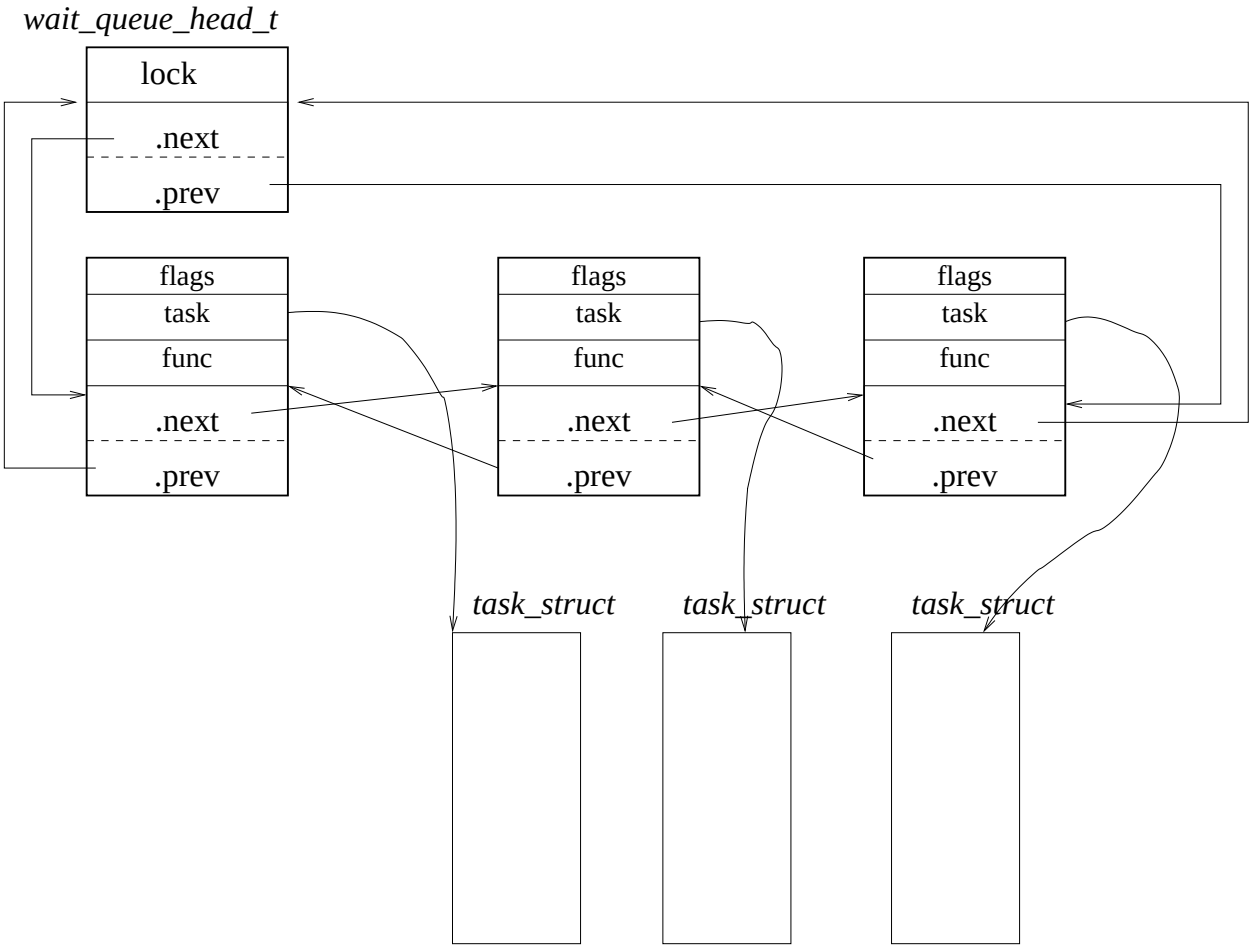
When a kernel task (in a synchronous situation) is about to perform an operation which could potentially be blocking, it prepares for this by creating a `wait_queue_t` entry (it is acceptable to make this a local variable...the kernel stack is a pretty safe place and control isn't about to leave the function in question) then inserting that entry into the wait queue head, then calling `schedule()` to relinquish the processor. There are two possibilities:

• Exclusive wait: The WQ_FLAG_EXCLUSIVE bit flag will be set in the `flags` field of the `wait_queue_t` entry. At most one exclusive process will be woken up at a time.

• Non-exclusive wait: WQ_FLAG_EXCLUSIVE is clear. All non-exclusive processes on the particular wait queue will be woken up.

Exclusive vs non-exclusive waiting

Exclusive waiters are normally inserted at the tail of the wait queue and non-exclusive waiters at the head. Therefore, if the queue should contain both types of waiters, all of the non-exclusive waiters are awoken first, then at most one exclusive waiter. This situation is actually not common: The choice of exclusive vs non-exclusive wait is made by the waiter at the time it goes to sleep based on whether it makes sense to wake up multiple waiters when the event arrives. If it is likely that only one task could proceed (e.g. waiting on a mutex lock) then the wait will be exclusive to avoid the **"thundering herd"** inefficiency (all tasks wake up, get scheduled, take up CPU time realizing that they have to go back to sleep again). If multiple tasks might be able to proceed (e.g. waiting for a pipe to drain to write more data) then non-exclusive waits make sense.



Wait interrupted by signal

In addition to exclusive vs. non-exclusive waiting, we can distinguish between interruptible and non-interruptible waits. It is useful to allow a signal to interrupt certain potentially long waits (e.g. waiting for a network message to arrive, waiting for a keyboard character). As we shall see, there is cleanup involved when this happens, so the majority of waits within the kernel are non-interruptible. A non-interruptible wait *should* be quick, but sometimes the event takes longer than expected. E.g. most disk I/O operations are non-interruptible. If the disk operation hangs, either because the media has been removed, or there is an I/O error that is being retried, this is manifested as a "freeze" of the process that can not be killed, even with SIGKILL! We hope that the disk driver layers are coded such that eventually the operation will time out and return a "failure" event instead of a "complete" event, which will also wake up the sleeper.

When the task puts itself on a wait queue, it sets its `state` to `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE` to indicate that it is sleeping. Either state prevents the scheduler from placing the task on a run queue, until it is woken up. The

TASK_INTERRUPTIBLE state allows the arrival of a signal to wake up the task too.

Wait queue example

```
// From /usr/src/linux/include/linux/list.h
// self-circular initialization
#define LIST_HEAD_INIT(name) { &(name), &(name) }


// From /usr/src/linux/include/linux/wait.h
// Following uses C-99 structure initialization syntax
#define DEFINE_WAIT(name) \\
         wait_queue_t name = {\\
                  .task=current,\\
                  .func=autoremove_wake_function,\\
                  .task_list=LIST_HEAD_INIT((name).task_list)\\
         }

// The following macro is defined in /usr/src/linux/include/linux/wait.h
// the do { }while(0) construct is a preprocessor trick to make this
// work syntactically as if it were a function
#define __wait_event(wq, condition)                                     \
do {                                                                    \
        DEFINE_WAIT(__wait);                                            \
                                                                        \
        for (;;) {                                                      \
                prepare_to_wait(&wq, &__wait, TASK_UNINTERRUPTIBLE);    \
                if (condition)                                          \
                        break;                                          \
                schedule();                                             \
        }                                                               \
        finish_wait(&wq, &__wait);                                      \
} while (0)


// The following code is abstracted from /usr/src/linux/kernel/wait.c
// there is a similar routine for exclusive waiting
prepare_to_wait( wait_queue_head_t *q, wait_queue_t *wait, int state)
{
         wait->flags&=~WQ_FLAG_EXCLUSIVE;
         spin_lock(&q->lock);                           /* simplified */
         /* This might not be the first time through the loop above, */
         /* in which case the wait_queue entry is already enqueued */
         if (list_empty(&wait->task_list))     /* skip if already enqueued*/
         {
                  list_add(&wait->task_list,&q->task_list);
         }
         // State will be either TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE
         // Either one keeps  this task off the run queue
         current->state=state;
         spin_unlock(&q->lock);
```

```
}

finish_wait(wait_queue_head_t *q, wait_queue_t *wait)
{
          __set_current_state(TASK_RUNNING);    // Basically redundant
          // The next passage basically takes the wait queue entry
          // "wait" and unlinks it from the CLL.  On a multiprocessor
          // system, another processor may also be in the process of
          // waking up this same wait queue entry.
          // list_empty_careful checks that next points to itself
          // AND prev points to itself.
          if (!list_empty_careful(&wait->task_list)) {
                    spin_lock(&q->lock);
                    list_del_init((&wait->task_list);     //remove from list
                    spin_unlock(&q->lock);
          }
}
```

This one of the typical methods as seen within the Linux kernel, however in other cases wait queues are manipulated directly by kernel routines. Unfortunately the Linux kernel is fairly sloppy and inconsistent with synchronization primitives.

Comparing this example with *condition variables* (Unit 6), the use of the mutex to guard against the lost wakeup problem is approached differently. Here in the Linux kernel, the calling task always places itself on the wait queue, before it has tested the condition, THEN it marks itself as being in a sleeping state (TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE in Linux kernel speak). Therefore it can not go to sleep without being on the wait queue. When we look at the wakeup routine below, we'll see that it examines all of the tasks on the given wait queue under the protection of the wait queue's mutex. Therefore no other task can be in the process of inserting itself into the wait queue while the wakeup is happening.

If the waker pulls the wait queue entry and wakes up the sleeper between the time that the sleeper called `prepare_to_wait` and when it called `schedule()`, there will be no missed wakeup. The wait queue lock is not released by the sleeper until it has changed its task state. A waker would gain the lock thereafter and set the task state back to TASK_RUNNING. Thus when the sleeper calls `schedule()` is is already awake and nothing happens.

Waking Up

Once a process puts itself to sleep, it can not be scheduled again until it is woken up. Therefore it is always another task (or an interrupt handler) which wakes up the sleeping process.

```
/*From /usr/src/linux/kernel/sched.c */
/* and /usr/src/linux/kernel/wait.c */
```

```
/* When this is called, the mutex spin lock q->lock has already been grabbed */

static void __wake_up_common(wait_queue_head_t *q, unsigned int mode,
                             int nr_exclusive, int sync, void *key)
{
        struct list_head *tmp, *next;

          // Iterate over wait queue CLL with variable tmp
        list_for_each_safe(tmp, next, &q->task_list) {
                wait_queue_t *curr;
                unsigned flags;
                   // Get back to start of wait queue entry
                curr = list_entry(tmp, wait_queue_t, task_list);
                flags = curr->flags;
                if (curr->func(curr, mode, sync, key) &&
                    (flags & WQ_FLAG_EXCLUSIVE) &&
                    !--nr_exclusive)
                        break;
        }
}

autoremove_wake_function(wait_queue *t, unsigned mode, int sync, void *key)
{
          try_to_wake_up(t->task,mode, wake_flags);
          list_del_init(&wait->task_list);
}

try_to_wake_up(struct task_struct *p,unsigned int state, int wake_flags)
{
          p->state=TASK_WAKING;
          rq=orig_rq=task_rq_lock(p,&flags);     // Get original run queue
          //Execute scheduler class specific hook
          p->sched_class->task_waking(rq,p);
          //Potentially move to different CPU
        cpu = select_task_rq(p, SD_BALANCE_WAKE, wake_flags);
        if (cpu != orig_cpu)
                set_task_cpu(p, cpu);
        rq = __task_rq_lock(p);
          /*..update scheduling statistics .. */
          /* Possibly: current->thread_info.flags|=TIF_NEED_RESCHED */
          activate_task(rq,p,1);                 // Mark task as runnable
          p->state = TASK_RUNNING;
}
```

This code iterates over the wait list. For each task, it calls the `func` function. Most often, this function was set to `autoremove_wake_function` by `DEFINE_WAIT`, which in turn calls `try_to_wake_up`. Despite the name, this function not only tries but generally succeeds at waking the task up, by setting its state to `TASK_RUNNING` and placing it on a run queue. Again, names are misleading. The state `TASK_RUNNING` is really a READY state. At some later time, the task will be selected by the scheduler and will become the `current` running task. (The `.func` element of the wait queue entry is there as a "hook" to allow situation-specific code to be executed when the wakeup takes

place.)

After the task has been woken up, `autoremove_wake_function` then removes the task from the wait list, still under the protection of the wait queue's spin lock mutex. In the case of an exclusive wait, no further tasks are woken up, but otherwise the list iteration continues and additional tasks are awakened.

One or more of the awakened tasks might have a (dynamic) priority greater than the current task (the waker-upper). If this is true, `try_to_wakeup` will set the TIF_NEED_RESCHED flag for the current task, which will be checked as the task returns to user mode, causing schedule() to be called. Thus a newly awakened task may pre-empt the current task upon return to user mode.

A system call with blocking

As a further example, we shall follow a system call which may involve the caller being put to sleep, specifically reading from a pipe which currently has no data in it. In this example, the generic __wait_event macro is not used, but the kernel code effectively does the same thing. The Linux kernel is generally inconsistent, and depending on when a particular part of the kernel was coded or revised, different synchronization mechanisms may be in play.

As discussed in the previous unit, the parameters to the system call (file descriptor, buffer address, count) are passed via registers. At the entry to `system_call`, the registers are saved on the kernel mode stack, in the same order in which arguments are normally pushed on the stack in a regular C program. Therefore, when the particular `sys_xxx` system call handler is invoked, it finds the parameters on the stack just as if it had been invoked as an ordinary function. The `__user` macro in the argument declaration for the read buffer pointer is there to remind us that the address is a user-mode address and is not trusted. We pick up our system call trace at `fs/read_write.c:sys_read`:

```
// From /usr/src/linux/fs/read_write.c

asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
{
          struct file *file;
          ssize_t ret ;
          loff_t pos;

                    // Next 3 lines are interpolation of several routines
          if (fd >= current->files->max_fds) return -EBADF;
          if (!(file=current->files->fdtable[fd])) return -EBADF;
          file->f_count++
          pos=file->f_pos;
          ret = vfs_read(file, buf, count, &pos);
          file->f_pos=pos;
          file->f_count--;
```

```
          return ret;
}

ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
        ssize_t ret;

        if (!(file->f_mode & FMODE_READ))               //open mode correct
                return -EBADF;
        if (!access_ok(VERIFY_WRITE, buf, count)))      //ptr to valid addr
                return -EFAULT;

          // rw_verify_area checks if the read request falls within
          // a mandatory record locking area of the file.  It also
          // does some range checks (e.g. negative position)
        ret = rw_verify_area(READ, file, pos, count);
          if (ret<0) return ret;
          count=ret;
          /*Most filesystems do not define op->read, but instead let the generic
            do_sync_read do the work by paging in the requested parts of file */
          if (file->f_op->read)
                    ret = file->f_op->read(file, buf, count, pos);  //VFS dispatch
          else
                    ret= do_sync_read(file, buf, count, pos);
          if (ret > 0) {
                    fsnotify_access(file->f_dentry);      //Hook for events
                    current->rchar += ret;                // Update stats
          }
          current->syscr++;                     // Update stats
        return ret;
}
```

As usual, the above code has been simplified somewhat from the actual Linux kernel sources. In particular, some complicated locking has been elided. The first thing done is to retrieve the `struct file` corresponding to the file descriptor. If the file descriptor is not valid, -EBADF is returned. The f_count field is incremented because an operation will be pending. Next, in the function `vfs_read`, other basic checks are performed. Was the file opened for reading (O_RDONLY or O_RDWR)? Is the read offset negative?

Finally, we are dispatched to the `read` method of the filesystem module which controls the file in question. This is performed via the `f_ops` structure of the `struct file`. Recall that the overall filesystem comprises one or more "mounted volumes," each of which may be of a different filesystem type. Each filesystem type has an associated module which provides methods for performing file operations such as read and write. (Un-named) pipes do not exist in the filesystem namespace, so there is a pseudo-filesystem module called `pipefs` which provides these methods when using pipe inodes. Our code walk-through winds up at `fs/pipe.c:pipe_read`:

```
ssize_t pipe_read(struct file *filp, char *buf, size_t count, loff_t *ppos)
{
        struct inode *inode = filp->f_dentry->d_inode;
```

```
        struct pipe_inode_info *info;
        ssize_t ret;
          int do_wakeup;

          do_wakeup=0;
          if (count==0) return 0;
          /* mutex_lock is a non-interruptible, blocking kernel mutex */
          mutex_lock(&inode->i_mutex);          /* Lock inode */
          info=inode->i_pipe;          /* Get private data */
          for(;;)
          {
            int bufs=info->nrbufs;
                    if (bufs > 0) {
                              /* ... This elided code copies the data from
                                        the pipe buffer(s) to user space, frees
                                        any buffers that have been completely copied,
                                        and adjusts info->nrbufs accordingly.

                                        ret+=number of bytes copied */
                              do_wakeup=1;                /* we have created room */
                              if (ret==count)             /* all req'd bytes */
                                        break;
                    }
                    else
                    {
                              /* Check for EOF condition (no writers ) */
                              if (info->writers==0)
                              {
                                        ret=0;
                                        break;
                              }
                              /* We have not read any data yet, so there is
                                 no way that we need to wake up a blocked writer,
                                 OK to sleep */
                              pipe_wait(inode);  //Releases and re-acquires mutex
                    }
                    if (signal_pending(current)) // Wait interrupted by sig
                    {
                               /* Only return ERESTARTSYS if we have read 0 bytes
                                * before waking up.  See text */
                              if (!ret) ret=-ERESTARTSYS;
                              break;
                    }
          }
          mutex_unlock(&inode->i_mutex);                /* Release inode lock */
          if (do_wakeup) wake_up_interruptible_sync(&inode->i_pipe->wait);
          if (ret>0) file_access(filp);                 /* update atime*/
          return ret;
}

/* The arguments to pipe_write look strange because all writes within
          the kernel are transformed into a "scatter/gather" form known
          as an "iovec" with nr_segs segments.  */
```

```
ssize_t pipe_write(struct file *filp, struct iovec *_iov,
                   unsigned long nr_segs, loff_t pos)
{
  ssize_t chars;
  size_t total_len,ret;
  struct pipe_inode_info *pipe;
  struct inode *inode=filp->f_dentry->d_inode;
  int do_wakeup;
          total_len=iov_length(iov,nr_segs);    //Compute total write req len
          do_wakeup=0;
          ret=0;
          mutex_lock(&inode->mutex);
          pipe=inode->i_pipe;
          if (!pipe->readers) {
                    send_sig(SIGPIPE,current,0); // Unit 4
                    ret= -EPIPE;
                    goto out;
          }
          for(;;) {
                    bufs=pipe->nrbufs; //How many buffers already in pipe
                    if (bufs<PIPE_BUFFERS)       {         //still room
                              /* Elided code creates a new buffer, copies the
                                        user data into it, and puts buffer into
                                        the FIFO list of buffers */
                              do_wakeup=1;
                              ret += chars;
                              total_len -= chars;
                              if (!total_len) break;       //done copying
                    }
                    //No more room
                    //eliding code for NONBLOCK operation
                    /*If signal woke us up and we came back around the
                              loop, still full, do an interrupted system call */
                    if (signal_pending(current)) {
                      /* If we have written some bytes, return that value.
                         This is a case where write to a pipe CAN return
                         a "short write"!.  Do interrupted syscall only
                         if we hadn't written any bytes before being interrupted */
                              if (!ret) ret= -ERESTARTSYS;
                              break;
                    }
                    if (do_wakeup) {
                              /* We have written at least some data, make sure
                                 to wake up readers blocked on empty pipe, BEFORE
                                 sleeping on full pipe condition */
                              wake_up_interruptible_sync(&pipe_wait);
                              do_wakeup=0;
                    }
                    pipe->waiting_writers++;
                    pipe_wait(pipe);             //Wait for more room
                    pipe->waiting_writers--;
          }
 out:
```

```
        mutex_unlock(&inode->i_mutex);
        if (do_wakeup)  wake_up_interruptible_sync(&pipe->wait);
        if (ret>0) file_update_time(filp);    //Update inode mtime
        return ret;
}



void pipe_wait(struct inode *inode)
{
        DEFINE_WAIT(wait);          /* declare wait entry */
        prepare_to_wait(&inode->i_pipe->wait,&wait,TASK_INTERRUPTIBLE);
        mutex_unlock(&inode->i_mutex);
        schedule();
        finish_wait(&inode->i_pipe->wait,&wait);
        mutex_lock(&inode->i_mutex);
}
```

There are two sleep/wakeup conditions with which to contend: reader sleeping until more bytes are written, and writer sleeping until there is space in the pipe buffer. Let us consider the first case only, as the second case is analogous.

The first thing which `pipe_read` does is obtain a blocking mutex lock on the in-core inode. This will prevent other operations such as read, write, stat, etc. on the inode while the pipe_read is running. Because we are in a system call, which is a synchronous situation, it is acceptable to use this potentially blocking operation.

If there are buffered data waiting on the pipe, they are copied to the user's buffer (we are not concerned with the mechanics in this unit) and `do_wakeup` is set, which will remind us later to wake up potential waiting writers.

If there are no buffered data, and there are still possible writers to the pipe, then the calling process must be put to sleep. An INTERRUPTIBLE sleep is chosen because it could be a long time before a writer process wakes us up, and the user should have the option of breaking out of the sleep with a signal. The sleep will also be non-exclusive, because once data are placed into the pipe, it is meaningful for multiple readers to proceed (they might all want just a little data and there is enough to go around).

`pipe_wait` handles the business of going to sleep. A wait queue entry is defined as a local variable with `DEFINE_WAIT` and placed on the inode's wait list with `prepare_to_wait`. Now the inode mutex can be released (and indeed *must* be released, to avoid sleeping with the mutex held, which would prevent any writers from ever accessing the inode). `schedule` is called, and one or more other tasks run. Note that this code does not use the generic `__wait_event` code seen above but instead rolls its own. This is fairly typical of the non-uniform coding style of the Linux kernel.

Note that the inode mutex is held while the wait queue entry is made. This protects against the "lost wakeup" problem although the locking is fairly coarse -- only one reader or writer can ever be working on this pipe inode at any given time.

At some later time, another task writes to the pipe. This writer task will call, in `pipe_write()` `wakeup_interruptible_sync`, which eventually brings us into the scheduler at `__wake_up_common`, seen above.

Therefore the reader process is eventually re-scheduled, and returns from pipe_wait(). Note the enclosing for(;;) loop which re-tests the condition (are there any data in the pipe?), because waking up does not necessarily mean the condition is now true.

Also note the check, after waking up, for pending signals. Posting a signal (which is not masked) to the waiting process here will cause the process to wake up, because the sleep is INTERRUPTIBLE. If some characters had already been read from the pipe before we had to sleep, then the read system call needs to return the number of characters read. However, if no characters were read, the system call will return ERESTARTSYS. See below under "Interrupted System Calls"

The reader (of these lecture notes, not the pipe) can also examine the pipe_read/pipe_write code above to see how a writer task, upon encountering a full pipe, goes to sleep and is awoken by a reader when the pipe is drained. Note how the pipe write atomicity (see Unit 4) is handled. Because pipe_write holds the mutex, no other writer's data can be interleaved. But, if there is not at least one full page-sized buffer available, or if the amount of data to write exceeds the available space in the pipe, the system call blocks after writing what it can. It is not an EXCLUSIVE wait so multiple writers can be blocked on the same pipe. When the pipe has room, they will all wake up but only one writer will win the mutex, and will get to write at least a 4K chunk of data atomically.

Sleep/Wakeup Summary

To summarize the sleeping and waking up process:
• Both the sleeper's code and the waker's code must agree on how the event is defined, and must both share a pre-declared wait queue.
SLEEPER:
• In the synchronous context of a system call or exception handler, allocates and initializes a wait queue entry
• The wait queue entry is added to the wait queue representing the event
• The task state is marked as TASK_INTERRUPTIBLE if the wait can be woken up by signal arrival, otherwise TASK_UNINTERRUPTIBLE.
• `schedule()` is called. The task is switched out, and because it is not in a runnable state, it will not get scheduled in again, until:
WAKER:
• The event arrives, either in a synchronous or an asynchronous context.
• The sleeping task is selected from the wait queue, and the wait queue entry is removed. For exclusive waits, only one task is selected, but for non-exclusive waits all waiting tasks are selected and removed from the wait queue.

• The task state is set to TASK_RUNNING and the task is placed on a run queue, making it eligible to be scheduled.
• If the woken task now has a "better" scheduling priority than the currently running task, the `TIF_NEED_RESCHED` flag is set on the current task, which subjects it to pre-emption by the woken task when execution returns to user level from the system call, fault or interrupt.
• The woken task is (at some later time) selected by the scheduler to run, and is switched in.

Interrupted System Calls / System Call Restart

In Linux kernel system call code, when a sleeping process is awoken not because the event has arrived, but because a signal is posted to that process (and that signal is not masked), that is known as an Interrupted System Call. This is a poorly documented aspect of UNIX systems programming. From the standpoint of the system call code, it can not continue because the event has not arrived and the signal needs to be delivered, therefore we need to return out of the system call and head back towards userland, at which point the signal can be delivered. The system call code returns one of the following error codes:
• EINTR: This system call can not be restarted. The error EINTR will be returned to the user.
• ERESTARTSYS: The system call will be restarted if the disposition of the signal in question is DFL or IGN. (Note: a signal which is being IGNored but which isn't masked will still cause a sleep to be woken up. Also note that while the DeFauLt action for many signals is process termination, whereupon system call restart is a moot point, there are a few where the default action is not termination). If there is a handler defined, the system call will only by restarted if the SA_RESTART flag is set for that signal number (e.g. through the `sigaction` system call). System call restart will happen after the handler returns. Most system calls use ERESTARTSYS.
• ERESTARTNOINTR: The system call will always be restarted, even if the handler does not have SA_RESTART set.
• ERESTARTNOHAND: The system call will be restarted if the signal disposition is DFL or IGN. If there is a handler, the system call will not be restarted, regardless of SA_RESTART, and upon completion of the handler EINTR will be returned from the system call.
• ERESTART_RESTARTBLOCK: This is a special case used for a few time-related system calls which require specific code to be executed prior to system call restart to adjust those time-related parameters. It forces a special `restart_syscall` system call to be executed upon resumption in userland to make this adjustments.

System call restart is effected by adjusting the user-mode registers prior to returning to user mode. The eip register (program counter) is decremented so that upon resumption of

userland code, the original `INT $0x80` opcode is executed again. The eax is reset to the original system call value. This is accomplished by modifying the saved user-mode registers which are on the kernel stack (see previous unit). Upon resumption of the user-mode program (and return from the signal handler if applicable) the system call will be re-called with the exact same parameters.

Process Termination

A process terminates either when it has received a fatal signal or it explicitly calls the `exit` system call. Let us look at the exit system call, ignoring some of the complexities introduced by threads and thread groups. `sys_exit` is a simple wrapper for `do_exit`. The latter can also be called during signal delivery.

```
asmlinkage long sys_exit(int error_code)
{
        do_exit((error_code&0xff)<<8);
}


void do_exit(long code)
{
        struct task_struct *tsk = current;

                    /* These are "can't happens" */
        if (unlikely(in_interrupt()))
                panic("Aiee, killing interrupt handler!");
        if (unlikely(tsk->pid==0))
                panic("Attempted to kill the idle task!");
        if (unlikely(tsk->pid == 1))
                panic("Attempted to kill init!");
          tracehook_report_exit(&code);         /* Hook for strace */
        tsk->flags |= PF_EXITING;
        exit_mm(tsk);                   /* Release virtual address space */
          exit_signals(tsk); /* Cleanup/reassign pending signals */
        exit_files(tsk);     /* Close open files */
        exit_fs(tsk);                   /* Leave cwd, etc. */
        exit_thread();                  /* Misc. cleanup */
          tsk->exit_code = code;        /* Exit cause code for wait() syscall */
          exit_notify(tsk);  /* Send signal to and/or wake-up parent */
        tsk->state= TASK_DEAD;
        schedule();
        BUG();              /* If control reaches here we are in trouble! */
}
```

Each of the helper routines can have further consequences. E.g. when `__exit_files` closes each file descriptor, if that drops the references to the `struct file` to 0, then that structure is de-allocated. If in turn there are no other `struct file` instances in the system pointing to the in-core inode, then the inode is closed. If the inode had been unlinked while open, its resources are freed, etc. After sending SIGCHLD to the parent

process, `exit_notify` places the exiting process in the `EXIT_ZOMBIE` state. Although most of the resources have been released, the `struct task` remains to hold the exit code and other statistics of the process's life, such as cpu time accumulated, for collection by a parent with one of the many variants of the `wait` system call. After this is done, the `struct task` is finally released and the process id is available for recycling. The zombie task is never scheduled, because nothing puts it onto a scheduler run queue. If somehow the last line is reached, it is evidence of a kernel bug!